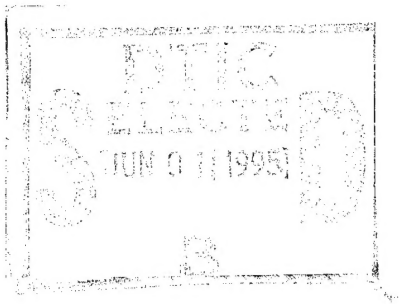
PARALLEL OCEAN MODELING USING GLENDA

Building 1103
Stennis Space Center
Mississippi 39529

Phone 601-688-5737
FAX 601-688-7072

Jerry L. Bickham, II

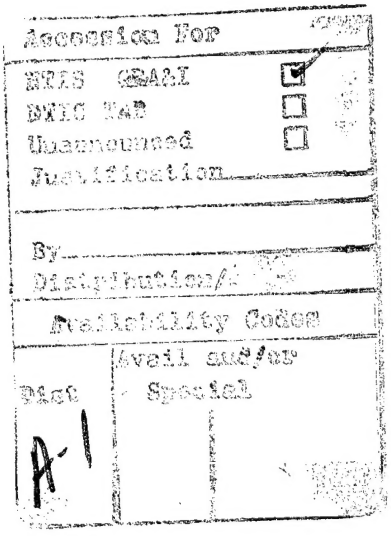


The University of
Southern Mississippi

TR-4/95

May 1995

The Center for Ocean & Atmospheric Modeling (COAM) is operated by The University of Southern Mississippi under sponsorship of the Department of the Navy, Office of the Chief of Naval Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.



University of Southern Mississippi

PARALLEL OCEAN MODELING USING GLENDA

by

Jerry Lynn Bickham, II

A Thesis
Submitted to the Graduate School
of the University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Master of Science

Approved:

Benjamin R. Seyfarth
Director

Mitchell Krell

Seemann Peppers

A. Dean Allen
Dean of the Graduate School

May 1995

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

In this document we describe how, with the use of Glenda, we were able to parallelize the ocean modeling program SWEM. After a description of Glenda and an introduction to the ocean model, we discuss how we parallelized the model. We first had to determine the array dependencies in the program in order to better understand the way we were going to handle data propagation. Once we had a grasp of how the data was to be propagated, there was a need to develop overlap definitions to help improve the readability of the code. But, of course, we still encountered problems that had to be solved. Therefore, we had to use a wide assortment of debugging techniques to overcome these problems. With our problems finally solved and the program now complete, we tested our new version of SWEM and gathered results.

2 Glenda

Glenda was developed in 1993 by Benjamin R. Seyfarth and graduate students (myself included) from The University of Southern Mississippi [11]. It was developed to provide both the basic capabilities of Linda[1] (a group of functions which simplify writing parallel programs) and the portability of PVM (Parallel Virtual Machine). The Glenda model was set up to closely parallel that of Linda - with a few exceptions. Glenda is built utilizing the PVM software system to provide the underlying communications [2]. PVM is a collection of functions that, like Linda, allow the user to make use of a multiprocessor system. Glenda supports most of the Linda operations with a few added capabilities to utilize PVM more fully [12].

2.1 PVM - Parallel Virtual Machine

PVM (Parallel Virtual Machine) is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational

[1] Linda is a registered trademark of Scientific Computing Associates.

resource. The individual computers may be shared- or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations, that may be interconnected by a variety of networks, such as ethernet, FDDI, etc. PVM support software executes on each machine in a user-configurable pool and presents a unified, general, and powerful computational environment of concurrent applications. User programs written in C or Fortran are provided access to PVM through the use of calls to PVM library routines for functions such as process initiation, message transmission and reception, and synchronization via barriers or rendezvous. Users may optionally control the execution location of specific application components. The PVM system transparently handles message routing, data conversion for incompatible architectures, and other tasks that are necessary for operation in a heterogeneous, network environment [6].

2.2 Linda and Its Use of Tuple Space

Linda programs communicate by inserting and retrieving tuples, or collections of data items referenced by a name, into a shared memory area referred to as "tuple space" [1] Therefore, before any further discussion of Glenda can proceed, the idea behind "tuple space" should be described. "Tuple space" is memory in one or more computers whose purpose is to serve as a temporary storage facility for data being transferred between processes. The data being transferred is grouped into collections called "tuples". Each tuple consists of a string, which serves as the tuple's identifier, and zero or more data items. The data can be scalar variables, array variables, or real numbers :

- ("Data", i, sum, A : 10, 5)

Here, the tuple's name is "Data". It consists of four data items. The variables $\mathbf{i}$ and $\mathbf{sum}$ are scalars. The variable $\mathbf{A}$ is an array of size 10. The final data item is the integer $\mathbf{5}$. When a process is ready to send a tuple to another process, the process to which it will send the tuple may not necessarily be ready to receive it. Thus, to prevent the sending process from having to wait, the tuple is temporarily stored in "tuple space" until the receiving process is ready to receive it [3][4].

2.3 Glenda Operations

Glenda is made up of five of the six Linda operations, as well as five other functions unique to Glenda[11]. Here are the Glenda functions.

- tid = gl_mytid()
- tid = gl_spawn (name, [hostname])

  We join Glenda by calling gl_mytid and use gl_spawn to start subprocesses. gl_mytid returns a PVM task id number and enrolls it into PVM. gl_spawn returns a PVM task id number for the spawned process.

- gl_out (name, ...)
- gl_in (name, ...)
- gl_inp (name, ...)
- gl_rd (name, ...)
- gl_rdp (name, ...)

  These are the five primary Glenda functions. Every tuple has a character string for its first component, followed by zero or more data items (represented above by ...). gl_out outputs a tuple into tuple space for other processes to retrieve using gl_in or gl_rd. gl_inp and gl_rdp are predicate versions of gl_in and gl_rd and only retrieve a tuple if a tuple is available.

- gl_outto (tid, name, ...)
- gl_into (name, ...)

  The functions outto and into were added to make use of PVM's multicast capability. gl_outto can be used along with a PVM task identifier to send a tuple directly to a task. gl_into must then be used to retrieve a tuple sent using gl_outto.

- gl_exit ( )

  To exit out of PVM and the Glenda tuple server, the function gl_exit must be used [12].

3 The Shallow Water Equation Model (SWEM)

SWEM is an acronym for Shallow Water Equation Model. It was developed by Katherine S. Hedstrom of Rutgers University. The SWEM code is derived from the external mode equations for the solution of the vertically-integrated flow which are part of the 3-D free surface models currently being developed by Prof. D. Haidvogel and his colleagues at Rutgers University. The principal attributes of SWEM are

- finite differencing
- Arakawa C-grid
- generalized boundary-fitted orthogonal coordinates
- option for masking out land areas

The generalized orthogonal coordinates were introduced to avoid the numerical inaccuracies of approximating the coastlines by a step-like function. In basin-wide applications, the complexity of the domain geometry makes it virtually impossible to adopt boundary fitted grids. However, curvilinear coordinates give the opportunity to concentrate fine spatial resolution in areas of higher interest and minimize the number of masked land points, indeed, reducing the computational cost of using Cartesian coordinates over the whole domain at the required fine resolution[10].

In general, ocean models describe the response of a variable density ocean to atmospheric momentum and heat forcing. This response can very simply be represented in terms of eigenmodes of a linearized system of equations. The zeroth mode is equivalent to the vertically-averaged component of the motion, known as the *barotropic* mode.

The higher modes are called *baroclinic* modes and are associated with the higher order components of the vertical density profile [5].

Ocean models usually make the hydrostatic approximation in which the pressure difference between two points on the same vertical line depends only on the weight as if the fluid were at rest. Under these assumptions, at any point the pressure forces depend on the thickness of the water column above that point, as well as on the vertical variations of the water density. Barotropic models neglect the 3-D structure of the density distribution and assume that the ocean is a homogeneous fluid. So, this relation holds if the horizontal dimensions of the ocean volume under consideration are much larger than the vertical dimensions, hence the *shallow water* designation [9].

3.1 Barotropic and Reduced Gravity Ocean Models

Barotropic models are interesting and important for several reasons. First of all, the free surface elevation couples directly to the barotropic mode. One of the important data fields used as part of the initialization and updating procedures for real-time ocean prediction are satellite altimeter measurements[2] of the free surface elevation. Thus, information from altimeters may first enter the ocean model through the barotropic mode.

The presence of free surface gravity waves represents a second important feature of barotropic models. Gravity waves are fast surface waves that propagate at a speed $c = \sqrt{gH}$, where g is the gravitational acceleration, and H is the depth of the ocean [9]. The simple explicit finite-difference schemes treating such waves are subject to severe time step limitations so that solution of the barotropic mode may lead to large CPU requirements. Therefore, it is important to study the solution of this system with efficient numerical schemes, before incorporating it into the general 3-D ocean models.

The pressure gradients associated with the free surface elevation are constant with depth. Thus, they form part of the zeroth mode or the vertically-averaged mode, and

[2] A radar altimeter measures the distance from the satellite to the surface of the ocean, and if the position of the satellite in space is known, this measurement allows the ocean surface deviations from the level corresponding to no motion to be inferred on time scales of a few days to years.

appear only in the barotropic mode equations. Consequently, the baroclinic system representing the higher-order modes has no surface elevation associated with it, and the corresponding surface boundary condition is that of a rigid lid.

A particular form of the baroclinic models are the so-called $reduced\ gravity$ models [5]. These traditionally assume a dynamically active upper layer of density ρ_1 which overlays a motionless, infinitely deep layer of density ρ_2. The corresponding mathematical equations are formally equivalent to the barotropic models with the gravitational acceleration g substituted by $g' = g\frac{(\rho_2-\rho_1)}{\rho_o}$, and the ocean depth, H, by the thickness of the upper layer, h. The reduced gravity acceleration, g', has a typical value of 2/3 cms^{-2}. Similarly, the fastest waves contained in the reduced gravity models travels at a speed $c = \sqrt{g'h}$, or about ten to one hundred times slower than the barotropic gravity waves[9].

So, as we form the model equations associated with SWEM, we will refer to barotropic model equations with the knowledge that these simple substitutions will result in the reduced gravity model formulations used in our simulation.

3.2 Model Equations for SWEM

The model equations for the barotropic component of a hydrostatic ocean are derived from the Navier-Stokes equations for incompressible flow on a rotating Earth. Here, they are presented in Cartesian coordinates with constant friction coefficients to simplify the arithmetical text.

$$\frac{\partial U}{\partial t} = +fV - gH\left(\frac{\partial \eta}{\partial x}\right) + (\tau_w - \tau_b)_x + A\nabla^2 U - \frac{\partial}{\partial x}(UU/H) - \frac{\partial}{\partial y}(UV/H), \tag{1}$$

$$\frac{\partial V}{\partial t} = +fU - gH\left(\frac{\partial \eta}{\partial y}\right) + (\tau_w - \tau_b)_y + A\nabla^2 V - \frac{\partial}{\partial x}(UV/H) - \frac{\partial}{\partial y}(VV/H), \tag{2}$$

$$\frac{\partial \eta}{\partial t} = -\left(\frac{\partial U}{\partial x} + \frac{\partial V}{\partial y}\right), \tag{3}$$

where

$\vec{U} = (U, V)$	–mass transports in the x- and y-directions, respectively
η	–free surface elevation
$\vec{\tau}_w = [(\tau_w)_x, (\tau_w)_y]$	–wind stress components
$\vec{\tau}_b = [(\tau_b)_x, (\tau_b)_y]$	–bottom stress components
$f = 2\Omega sin\phi$	–Coriolis parameter for latitude ϕ; Ω is the angular rotation rate of the Earth $7 \times 10^{-5} s^{-1}$
$H(x, y)$	–topography (bottom depth)
g	–acceleration due to gravity
∇^2	–Laplacian operator in horizontal coordinates x, y
A	–coefficient of lateral friction

Now we must choose our boundary conditions. The main criterion is whether a boundary is closed (like at a shore) or open (like at a strait where waves and currents can enter and exit the model region). At rigid walls (a coast, the ocean bottom, etc.) no flux of momentum is prescribed. That is, the cross boundary velocity is set to zero. Another condition that is often applied is setting the tangential velocity to zero at the lateral coastal boundary. The vanishing of the tangential velocity implies the existence of frictional boundary layers, because the velocity is brought to zero from the free stream value across a thin boundary layer; and in this layer, friction is important. At the ocean bottom, the effects of friction are represented by the bottom stress, τ_b. SWEM applies a linear drag coefficient, such that $\tau_{b_x} = C_d U$ and $\tau_{b_y} = C_d V$. At the sea surface, the wind stress represents the input of energy from the atmosphere [5].

Of course, having a closed boundary is not our only option. However, boundary conditions at open boundaries (such as a strait or a gulf) are more difficult to assign. In general, it is necessary to specify, either from observations and/or estimates, features that enter the domain and allow features to exit without generating disturbances inside the modeled region [9].

3.3 Numerical Schemes for SWEM

The system of equations (1)-(3) has both parabolic and hyperbolic properties, the former associated with the diffusion terms and the latter with the pressure gradients and nonlinear terms. Diffusion of momentum will lead to a parabolic partial differential equation. The coupling of the time derivatives to the pressure gradients will lead to a system that has hyperbolic characteristics.

The numerical schemes for solving time-dependent partial differential equations fall generally into two classes: explicit or implicit. The term "explicit" denotes a scheme where all terms on the right hand side (r.h.s.) of system (1)-(3) are evaluated at time steps n, n - 1, etc. So, at any time t^{n+1} the r.h.s.is known from previous steps. "Implicit", on the other hand, denotes a scheme where some of the terms on the r.h.s. are evaluated at time step t^{n+1} and, thus, we must transfer these terms to the left hand side (l.h.s.) of the equations and invert the corresponding coefficient matrix of the unknown variables at t^{n+1}.

Even with the recent advances in the numerical solution of partial differential equations using the implicit treatment, the different nature of new computer architectures and the increasing size of computer memories are leading to renewed consideration of the explicit treatment of the barotropic mode. As the number of vertical levels or layers increases, the fraction of computer time spent in solving the 2-D barotropic equations per level tends to decrease. Also, most explicit techniques are usually fully vectorizable and "parallelizable", giving strong competition to the implicit techniques on computers of this and the next decade [5]. Therefore, since SWEM uses explicit numerical schemes to solve the system (1)-(3), we will now concentrate on explicit time integration.

The finite difference analog of equations (1)-(3) follows from the finite difference expressions for the first and second spatial derivatives in the x- and y-directions. Denoting $f(x, y) = f(i\,dx, j\,dy) = f_{i,j}$, we have

$$\frac{dU}{dx} = (U_{i+1,j} - U_{i-1,j})/2\Delta x, \tag{4}$$

$$\frac{dV}{dy} = (V_{i,j+1} - V_{i,j-1})/2\Delta y, \tag{5}$$

$$\frac{d^2U}{dx^2} = (U_{i+1,j} + U_{i-1,j} - 2U_{i,j})/\Delta x^2, \tag{6}$$

$$\frac{d^2V}{dy^2} = (V_{i,j+1} + V_{i,j-1} - 2V_{i,j})/\Delta y^2. \tag{7}$$

In order to obey the stability conditions of the finite difference scheme, the terms of equations 4-5 are computed at $t = n$, and 6-7 are computed at $t = n - 1$. Similarly, the pressure and Coriolis terms are computed at $t = n$, while the bottom stress, τ_b, is computed at $t = n - 1$ [7].

By locating the variables on a staggered mesh called the C-grid, the pressure p and height h variables are located at the center of the mesh boxes, and the mass transports U and V at the center of the box boundaries facing the x and y directions, respectively.

The Arakawa C-grid (shown in figure 1) [7] is the grid model being used. The thick outer line shows the position of the model boundary. The points inside the boundary are those which are advanced in time using the model physics. The points on the boundary and those on the outside must be supplied by the boundary conditions.

The two-dimensional model fields are carried in three-dimensional arrays where the third array index refers to the two time levels (old and new). The integers $\mathbf{i}$ and $\mathbf{j}$ are used throughout the model to index the two spatial dimensions:

- $\mathbf{i}$ Index variable for the ξ direction.
- $\mathbf{j}$ Index variable for the η direction.

The range of ξ is 1 to $\mathbf{L}$ and the range of η is 1 to $\mathbf{M}$ [7].

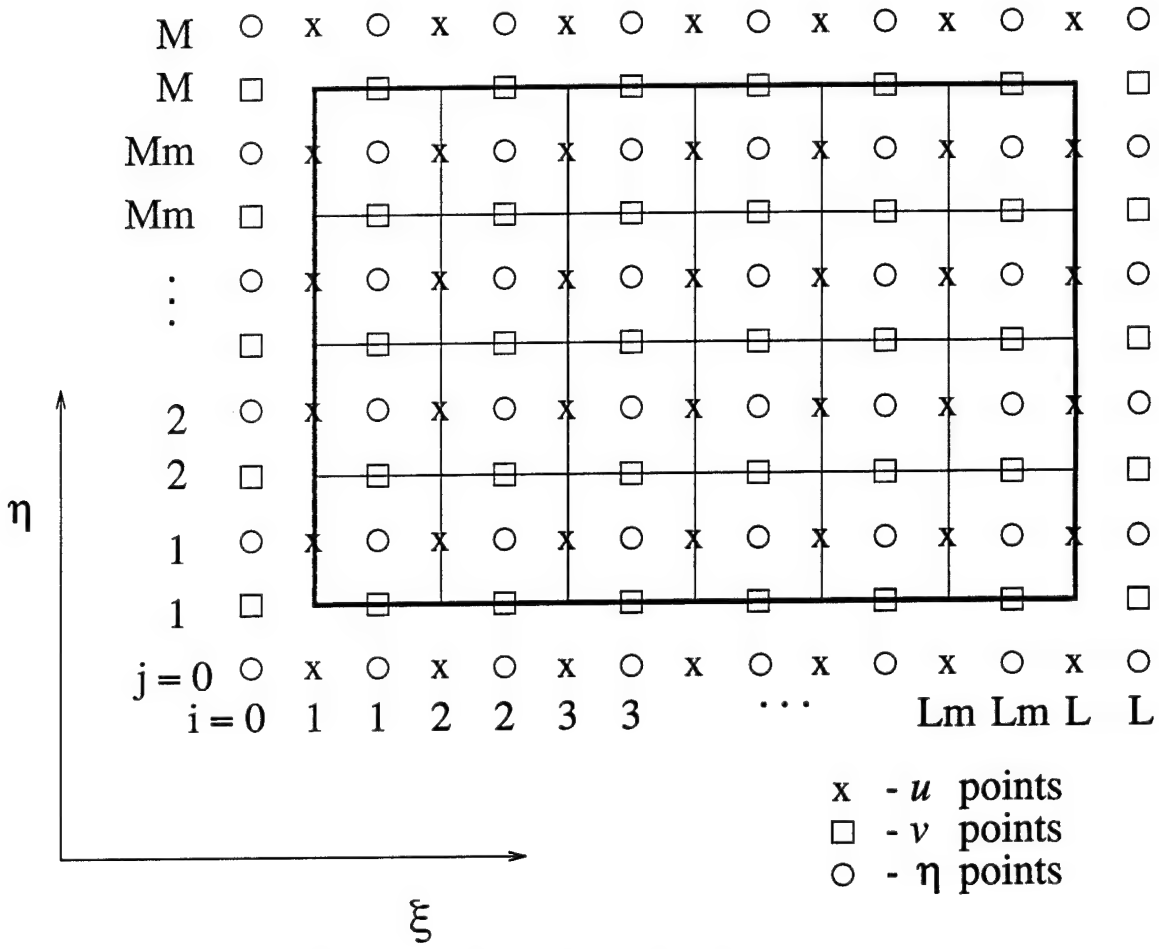


Figure 1: The Arakawa C-grid.

4 The Parallelization Process

The process of parallelizing the SWEM code consisted of many steps. First was determining the dependencies between adjacent array cells within the primary arrays. Next, we had to determine both how and where the data was to be propagated between workers. We then had to derive overlap definitions to define the amount of data to be shared between workers. Once initial coding was completed, many unforeseen problems resulted in which a variety of debugging methods were devised to aid in solving them. Lastly, when

all of the coding was completed, we tested our parallel model for speedup.

4.1 Parallelization Strategy for SWEM

The SWEM code is comprised of three major parts. The first part is the initialization phase where all of the primary variables are given starting values. Part two consists of three nested loops which do the actual calculations. The final part is the output phase where all necessary data is recorded.

Our first objective was to determine which of these parts lent themselves to be parallelized. Since the first and last parts made up only a small percentage of the computation time, they were left essentially unchanged. The bulk of the code's work was being done by the three loops, and it was this portion of the code that was to be parallelized.

The primary data consists of three 3-dimensional arrays - **ubar**, **vbar**, and **zeta** - along with a multitude of scalars and 2-dimensional arrays used for intermediate calculations. The 3-dimensional arrays were of size Row_length * Column_length * 2, with the 2 in the third dimension representing the old and the new values for the two dimensional components. For example, **ubar**(i,j,kold) represents the old value for the two dimensional **ubar**(i,j), while **ubar**(i,j,knew) represents the new value. As computations with these arrays proceed, the data for **knew** is usually overwritten with the data from the previous time step (t^{n-1}), and the values of **kold** and **knew** are switched.

The parallel model we incorporated into SWEM was the master-worker scheme, where each worker would do exactly the same calculations, but on a different portion of the data [4]. This scheme not only seemed to be best suited for SWEM's structure of calculation, but also was well suited for SWEM's structure of data. The data - primarily two and three dimensional arrays - could be divided among all of the workers, and it was this division that determined how much calculation time would be spent by each worker. Each worker would basically receive its data from the master program, execute the three nested loops, and return its results back to the master program.

4.2 Determining Array Dependencies

Many of SWEM's calculations consist of averages or differences among array cells which are nearest neighbors. Some even depended on next nearest neighbors. From this, it was evident that each worker would have to share data with an adjoining worker. Therefore, we had to determine how much data was to be shared between the workers to ensure correct calculations. One possible method of doing this would be to painstakingly read every line of code to determine which arrays depended on which other arrays. Seeing that there were on the order of fifty arrays to check, doing this by hand was out of the question.

To help speed things along, we wrote a simple array dependency program which scanned a program and determined each array's dependency on other arrays. Before using this array dependency program, we first had to insert into each subroutine blocks of commented lines consisting of an array name, its indices, and any arrays that it depended on in a single calculation. Using this program we rapidly determined that dependencies spanning two rows and columns existed for the three main arrays based on inputs and outputs of complete subroutines. For example, in this calculation

```
tmp2(i,j) = mDon_r(i,j,krhs) *
            (ubar(i+1,j,krhs) - ubar(i,j,krhs))
```

the array **tmp2** depends on two arrays - **mDon_r** and **ubar**. However, calculation of an array cell of **tmp2** depends on **mDon_r** and two elements of **ubar**. This represents a neighbor dependency along each row of **ubar**. Also, we do not know exactly which arrays these two depend on. To determine this, we would place these comments below this statement.

```
c
cdep  tmp2(i,j) ~ mDon_r(0,0,0) ubar(0:1,0,0)
c
```

The 'dep' tells the array dependency program that this comment line is to be processed. Next is the array to be checked along with its dimensions. The '~' represents

'depends on' and was used to help the programmer distinguish the line from the original assignment statement. Following the '$\sim$' symbol are the arrays that **tmp2** depends on in the calculation. Inside the parentheses, the numbers represent the neighbor dependencies within an array. Essentially, a descriptor of the form

```
- number_of_left_neighbors : number_of_right_neighbors
```

is used for each dimension, unless there are no neighbor dependencies at all, in which case a '0' is used. For example, (0,0,0) represents no neighbor dependencies in three dimensions. (0:1,0,0) represents a right neighbor dependency in each row, and no neighbor dependencies in the other two dimensions. (0,-1:1,0) represents a top and bottom dependency in each column, and no neighbor dependencies in the other two dimensions. When the dependency program is run on our example, the following output is produced:

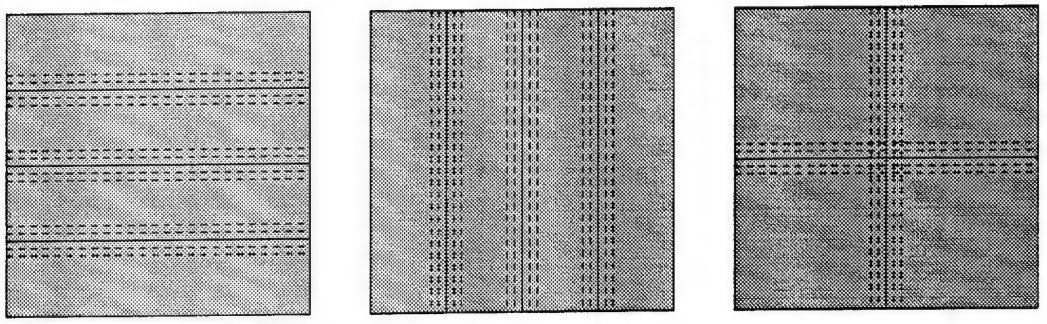
```
c
cdep  tmp2(i,j) ~ mDon_r(0,0,0) ubar(0:1,0,0)
c     tmp2(i,j) ~
c               mDon_r(i,j,k)
c               ubar(i:i+1,j,k)
c
```

4.3 Data Propagation in SWEM

After examining all of the subroutines called within the three nested loops with the array dependency program, we determined that the largest neighbor dependency was a two neighbor dependency in both the first and second dimensions. We then had to decide which method of dividing the data we would use, how we would propagate data from the master to the workers, and how we would propagate data between workers.

4.3.1 Dividing the data

We essentially had three choices of how we could divide the data, as shown in Figure 2: 1) divide the data by rows, 2) divide the data by columns, or 3) divide the data by



Division by rows Division by columns Division by rows and columns

Figure 2: Three choices for dividing the data.

both rows and columns. Dividing the data by rows and dividing the data by columns are essentially the same. The only difference is the way you visualize the data.

However, since the program was written in FORTRAN, which uses column major array indexing, we decided that division by rows would make for more efficient data transmission between workers. It would also make for easier coding of the data transmission subroutines to be written using Glenda. With division by columns, the data being sent would not be contiguous and would require multiple sends to transport the data from one process to another. On the other hand, with division by rows, the data could be sent in contiguous blocks which would require only one sending call.

We also considered dividing the data by both rows and columns, but this too had its problems. First of all, more divisions meant that more sending calls would be needed to send all of the data. This would increase communication times and decrease efficiency. But most importantly, even though division by both rows and columns gives us less total data to have to work with, the difference between this method and division by rows can be negligible. To illustrate this, consider an $n \times n$ array of double precision, floating point numbers divided between p^2 workers. With division by rows only, we have for the amount of data in bytes

$$Amt\_of\_Data = n * [n + 4 * (p^2 - 1)] * 8 \qquad (8)$$

Number of Workers	4	16	64
Division by Rows	89600	128000	281600
Division by Rows and Columns	86528	100352	131072

Table 1: Amount of data (in bytes) being processed per worker.

With division by rows and columns, we have

$$Amt\_of\_Data = [n + 4 * (p - 1)]^2 * 8 \quad (9)$$

Suppose the array we are working with is 100 $\times$ 100, and we divide it between four workers. With division by rows only we have 89600 bytes. With division by rows and columns we have 86528 bytes. Thus, there is only about a three percent difference in the amount of data being worked on by all of the workers. Table 1 shows how the amount of data being processed (in bytes) varies as we increase the level of parallelization. As seen in the table, as the number of CPUs increases, the difference between division by rows and division by rows and columns increases significantly. Therefore, if we run the simulation with more processors, it would be advisable to divide the data by both rows and columns for increased efficiency.

4.3.2 Propagation from master to worker

Each worker is responsible for computing results for a range of rows of the major arrays. After reading the input data, the master program uses the `gl_outto` operation to send the appropriate sections of these arrays to each worker [11]. At this point the master and the workers simultaneously complete the initialization process by computing scalars and arrays derived directly from the input data.

Once we decided on dividing the data by rows, we could now begin to write the code for the propagation of data between processes. Transfer of data between the master process and its worker processes was trivial. We simply divided the data into blocks and

sent each block to the appropriate worker using a subroutine written in C using Glenda to transfer the data between processes. Each worker will be responsible for a particular block of data in each array. To send the data arrays from the master to the workers, the call

```
gl_outto(worker_tids[i],"3D_arrays",
         ubar + ubar_offset : ubar_size,
         vbar + vbar_offset : vbar_size,
         zeta + zeta_offset : zeta_size);
```

was used. The array "worker_tids[i]" contains the task id number for the i'th worker process. The string "3D_arrays" is the name of the data tuple being sent. Each array being sent is represented by

- array name + array offset for worker i : block size for worker i

To receive the data arrays that were sent from the master to the workers, the call

```
gl_into("3D_arrays",
         ? ubar :len,? vbar :len,? zeta :len);
```

will be used. Again, the string "3D_arrays" is the name of the data tuple being received. The "?" tells Glenda to place the corresponding data item in the matching tuple into the variable following the "?". If a data item being sent to **gl_into** is an array, then a variable is used to receive the size of the array being sent. Here, the sizes of the arrays was not needed, so the sizes were received in a dummy variable **len**. If some of the data being sent were scalars, then the size and offset would not be needed in the **gl_outto** call, thus removing the need for a size variable in the **gl_into** call.

4.3.3 Propagation between workers

The SWEM workers were each assigned a range of rows to compute new values within the inner loops. In an example with 97 rows and three workers we divided the data as illustrated in Figure 3.

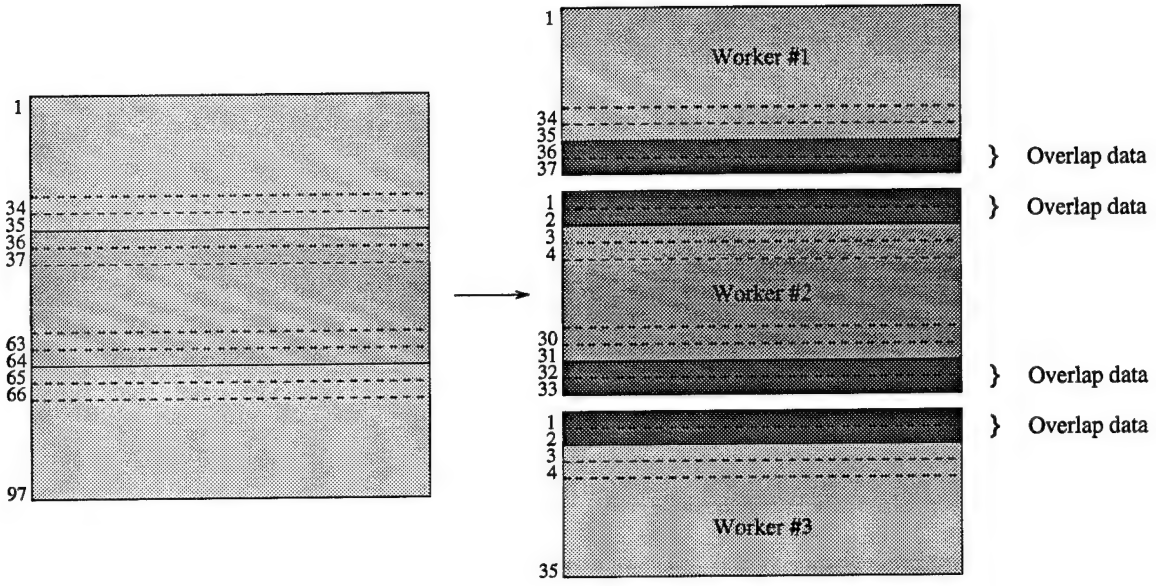


Figure 3: Division of data with 97 rows among three workers.

Worker 1 computes new data for rows 1 through 35, worker 2 computes new data for rows 36 through 64, while worker 3 computes new data for rows 65 through 97. The calculations for worker 1 require data for 2 additional rows: 36 and 37. Also the calculations for worker 2 require data for rows 34 and 35. After computing new data for rows 36 and 37, worker 2 needs to send this data to worker 1 to prepare it for the next iteration. At the same time worker 1 needs to send its new values for rows 34 and 35 to worker 2 to prepare it for the next iteration. A similar process occurs between workers 2 and 3.

4.3.4 Overlap definitions

During the process of writing the code for the propagation routines, it soon became evident that there was a need for some easy way to allow myself to write "readable" code. Each time there was a call to **gl_outto** or **gl_into**, the size of the overlap regions were needed for each array. Also needed was the offset required to get to the last two rows of data in each array (the overlap with the next worker). And finally, in order to know how many rows of overlap to send to an adjacent worker, you also needed to know whether

the array's row indices began with 1 or 0.

To solve this problem, we developed fixed equations that defined exactly which data region was to be sent to adjoining workers. These equations would define the offset and size for each boundary region to be propagated. They are as follows:

```
worker_0 = (worker == 0) ? 2 : 0
if worker < (num_workers - 1)
     offset =(num_rows - worker_0) * array_width
     size   =((array_start == 2) ? 1:2)*array_width
if worker > 0
     offset =(3 - array_start) * array_width
     size   =(2 - array_adj) * array_width
```

Here, **worker_0** is defined to be equal to 2 if the worker in question is the worker designated as the first worker (worker 0). Otherwise, its equal to 0. This variable is used to determine how many rows from the end of the array the overlap block begins. If the worker in question is not the last worker, then it must send its last two (or one) rows that it is responsible for calculating to the next worker. In this case, **offset** and **size** are as follows: **offset** defines where in the array the overlap block begins. **num_rows** defines how many rows of the array the worker is working with. **array_width** defines the number of columns in the array. **size** defines how many array elements are in the overlap block. **array_start** is the starting row index for the array. If it is equal to 2, the size of the overlap block is only **1 * array_width**. If it is equal to 0 or 1, the overlap block is equal to **2 * array_width**.

If the worker in question is not the first worker, it must send its first two rows that it is responsible for calculating to the previous worker. The **offset** skips past the first two (or three) rows of overlap data to get to the correct block. The **size** is determined by **array_adj** and, of course, the **array_width**. If the array in question has a beginning row index of 2, it has an array adjustment - **array_adj** - of 1. If it has a beginning row index of 0 or 1, then **array_adj** is equal to 0. Thus, the **size** will be equal to **2 * array_width**, unless the array has a beginning row index of 2.

Here is the code required to send the propagated data from one worker to another.

```
worker_0 = (*worker == 0) ? 2 : 0;
if( *worker < (*num_workers - 1) )
{
  zeta_offset =(*num_rows - worker_0)* zeta_width;
  zeta_size   =((zeta_start == 2) ? 1 : 2)* zeta_width;
  ubar_offset =(*num_rows - worker_0) * ubar_width;
  ubar_size   =((ubar_start == 2) ? 1 : 2)* ubar_width;
  vbar_offset =(*num_rows - worker_0) * vbar_width;
  vbar_size   =((vbar_start == 2) ? 1 : 2)* vbar_width

  gl_outto(worker_tids[*worker+1],"bound_bottom",
           zeta+zeta_offset : zeta_size,
           ubar+ubar_offset : ubar_size,
           vbar+vbar_offset : vbar_size);
}

if( *worker > 0 )
{
  zeta_offset =(3 - zeta_start)* zeta_width;
  zeta_size   =(2 - zeta_adj)* zeta_width;
  ubar_offset =(3 - ubar_start)* ubar_width;
  ubar_size   =(2 - ubar_adj)* ubar_width;
  vbar_offset =(3 - vbar_start)* vbar_width;
  vbar_size   =(2 - vbar_adj)* vbar_width;

  gl_outto(worker_tids[*worker - 1],"bound_top",
           zeta+zeta_offset : zeta_size,
           ubar+ubar_offset : ubar_size,
```

```
            vbar+vbar_offset : vbar_size);
}
```

Notice that for some workers, both **if** conditions will be true. These workers are responsible for interior blocks which have overlapping regions at both the beginning of the array and the end of the array.

In order to receive these blocks of data, each worker must also have similar definitions for the offsets so that the blocks of data are placed in the correct locations within the array. They are as follows:

```
worker_0 =(worker == 0)? num_rows + 1 :num_rows + 3;
if worker < (num_workers - 1)
     offset =(worker_0 - array_start)* array_width
if worker > 0
        offset = 0
```

Here, **worker_0** defines how many rows to skip in order to locate the correct location within an array to place the received block. If the array in question has its row index begin with **array_start** and the worker in question is not the last worker, then that value is subtracted from **worker_0** to get the correct offset for that particular array. If the worker in question is not the first worker, then the offset is 0 for all arrays. An example of the code designed to receive the blocks is as follows:

```
worker_0 =(*worker == 0)? *num_rows + 1 :*num_rows + 3
if(*worker < (*num_workers - 1) )
{
  zeta_offset = (worker_0 - zeta_start) * zeta_width;
  ubar_offset = (worker_0 - ubar_start) * ubar_width;
  vbar_offset = (worker_0 - vbar_start) * vbar_width
  gl_into("bound_top",
          ? zeta+zeta_offset : zeta_size,
```

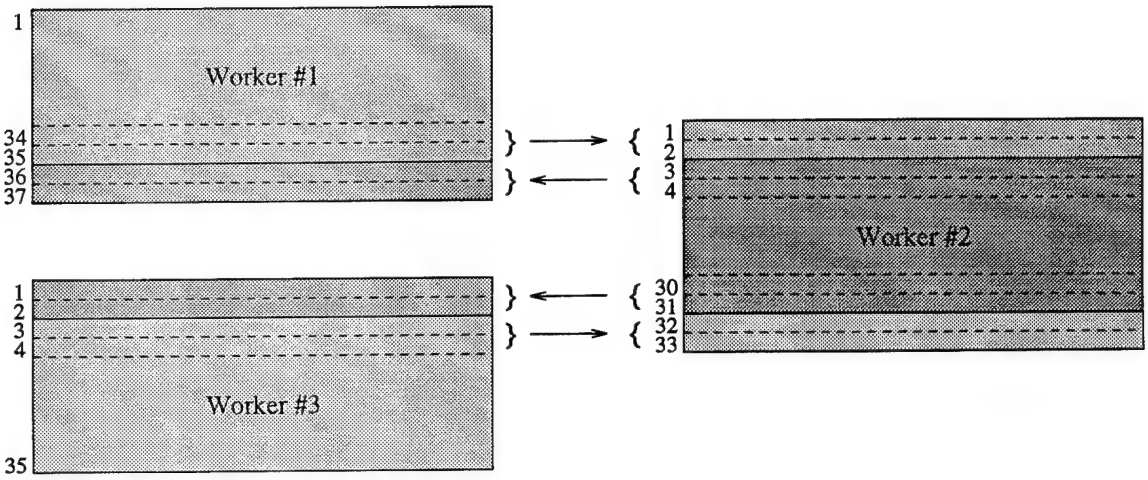


Figure 4: Propagation of data between workers.

```
            ? ubar+ubar_offset : ubar_size,
            ? vbar+vbar_offset : vbar_size);
}
if(*worker > 0)
{
  zeta_offset = 0;
  ubar_offset = 0;
  vbar_offset = 0;
  gl_into("bound_bottom",
            ? zeta+zeta_offset : zeta_size,
            ? ubar+ubar_offset : ubar_size,
            ? vbar+vbar_offset : vbar_size);
}
```

Again, if a worker is not the first or last worker, both **if** conditions hold. Therefore, each of these workers will require two receives. The procedure for propagation of data between workers is demonstrated in Figure 4.

4.4 Problems Encountered

As already seen, many problems were encountered during the process of attempting to parallelize the SWEM code. There were many obstacles standing in our way at the very start. Some of the problems we encountered were simple problems that were rather easy to solve, while others ended up costing us valuable time and effort.

One major problem was with the arrays themselves. In the initial stages of our research, our knowledge of both ocean dynamics and the mathematical basis behind the model was rather limited. Many of the array names gave little indication as to their purpose. Names such as **omn_u**, **mDon_r**, and **on_u** told us very little about their meaning. Also, many arrays had names that differed in only one or two characters. Arrays such as **nDom_r** and **mDon_r**, **ubar** and **vbar**, and **dndx** and **dmde** could cause the program to produce incorrect data if they happened to accidentally be exchanged for one another. These types of errors are extremely difficult to find, and can cause many long delays in the coding process.

Another problem we had with the arrays was the way they were defined. Each array was given a starting index and an ending index for each dimension. Unfortunately, these starting and ending indices were usually different from one array to another. These differences caused us to lose hours of valuable time that could have been spent programming rather than debugging. If an array size or an array's starting index was mistakingly given the wrong value, we could end up placing values past the end of an array and writing over half of our data in memory in the process. Once this happens, a close inspection of all of the code is usually required to find out where it went wrong.

Yet another problem we had to deal with was how to handle masking arrays used in the original SWEM code to mask out any land areas that were in the particular grid being modeled.[3] The SWEM masking arrays consist of single-dimension arrays of indices which define the locations of land areas within the study grid. These arrays are used within SWEM to reference the land grid points while treating the various two-dimensional data arrays as single-dimension arrays beginning with index 1. This makes the masking

[3]This was the only part of the original code that had to be adjusted.

arrays consist of index values which require careful interpretation to determine which indices refer to the sub-grid processed by a particular worker.

We wrote a subroutine that modifies any masking arrays being used, and limits the masks to only those which refer to data contained within a particular worker. It also subtracts an offset from its indices to make them correct for the worker's sub-arrays. The amount of offset within the worker is dependent upon whether this worker is the first worker, the widths of the arrays, and whether the arrays being adjusted start at row 0, 1, or 2 (this problem was mentioned earlier).

4.5 Debugging Methods Used

With all of the problems we encountered, there was a need for an easy way to know whether or not the data being calculated by the parallel model was correct. One way to do this was to compare the final output files of the parallel version of SWEM with those of the sequential version. However, if there were any differences in output, there was no way to know where the error occured. Therefore, there was a need to be able to locate errors at points within the code. To accomplish this task, we decided to run the sequential version of SWEM and the parallel version of SWEM concurrently within the same program. The part of the program being parallelized was left in the main program and was executed in step with the worker processes that were doing the parallel part. At strategic points within the master process and the worker processes, we would suspend calculations and send data from the parallel part to the sequential part of the program for comparison. Using generic send and receive routines, we would send over one array at a time and use a generic compare routine to compare it with its sequential counterpart. If they were the same, we would try another array until we found the one causing the error. Of course, before we could use these routines, they had to be properly debugged themselves. The problems of using the correct array sizes, and using the proper starting index caused us a little grief in the initial stages of debugging the code. However, once the debugging subroutines were debugged themselves, they saved us days (if not weeks) of debugging time.

	Time	Speedup	Efficiency
Sequential	3:54:41	1.00	
Pfa (2 workers)	2:06:08	1.86	.93
Pfa (3 workers)	1:34:36	2.48	.83
Glenda (2 workers)	2:16:25	1.72	.86
Glenda (3 workers)	1:48:27	2.16	.72

Table 2: Timings for a 240 day simulation.

5 Testing the Parallel Version of SWEM

For our test, we used a three CPU SGI 340-GTXB, operated by Mississippi State University Center for Air Sea Technology (CAST). We were given sole use of the computer for testing purposes. As part of the test, we ran the original sequential version of SWEM to serve as a basis for comparison. Using this time, we will be able to determine our parallel version's speedup. For a more challenging test, we also compiled the original SWEM code using the SGI MIPS FORTRAN 77 "-pfa" option [8]. This option runs the **pfa** (POWER Fortran Accelerator^{TM}) preprocessor to automatically discover parallelism in the source code. It also enables the multiprocessing directives for the SGI 340-GTXB. This timing gives us a "yardstick" upon which we can measure our parallel version.

The timings were made for 240 day simulations.

Our results are shown in Table 2. These efficiency values are not particularly high. The primary reason for this is that the array sizes are relatively small. In the case of the SGI compiler it requires moderately long loops to achieve high efficiency [8]. In the Glenda version the small width of the rows means that there is a relatively small amount of time spent computing between communication steps. With larger data sets, we predict that both the SGI Fortran compiler and the Glenda code would achieve higher efficiencies. However, it is unlikely that a message-passing solution will out-perform the tightly-coupled code produced by the compiler.

In a similar test, we compared the Glenda version with a version of SWEM using PVM

to handle the communications. As expected, the PVM version was slightly more efficient than the Glenda version - but only by about two percent. This test was based on a 48 day simulation with only two workers. Due to hardware limitations and time constraints at the time of testing, we were not able to test a 240 day simulation or test with more than two workers. However, since we have had similar comparative results using Glenda and PVM, we feel that this test was an accurate comparison.

6 Conclusion and Future Research

We have shown that Glenda can be used to efficiently parallelize an ocean modeling program. While the efficiency is lower than the SGI compiler's, the Glenda version of SWEM is portable to a variety of environments. We have illustrated how we divided the work among the workers and resolved the resulting communication problem.

The decision to divide the arrays by rows resulted in an efficient solution in our test environment. Our tests were done with a moderately small data set. We predict that with larger-sized data sets, this version of SWEM will be efficient up to perhaps eight workers. It is clear that division by rows limits the scalability and at some point it would be necessary to divide the data by rows and columns.

Our research has shown that Glenda shows promise for working with ocean models. We plan to extend this effort by testing the parallel SWEM in new environments. In particular we are interested in determining how efficiently we can utilize massively-parallel machines.

Acknowledgments

This author is greatly indebted to many people without whom this research effort would not have been possible. A special thanks to my committee chairman Dr. Benjamin R. Seyfarth (The University of Southern Mississippi) and its members Dr. Germana Peggion (COAM) and Dr. Mitch Krell (The University of Southern Mississippi) for their continuing guidance and support of this research.

This work was supported by the Office of Naval Research (under Grant # N00014-92-

J-4112), the National Aeronautics and Space Administration (NASA), and by the Center for Ocean & Atmospheric Modeling (COAM), Stennis Space Center, Mississippi. The author would also like to thank the Department of Computer Science and Statistics and COAM for not only being given the opportunity to pursue this research, but also for being allowed to participate in this Master's program and broaden his horizons.

References

[1] Robert D. Bjornson, "Linda On Distributed Memory Systems", *UMI Dissertation Services*, Yale University, May. 1993.

[2] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek and Vaidy Sunderam, "A User's Guide to PVM", *Oak Ridge National Laboratory*, Oak Ridge, TN, Jul. 1991.

[3] Nicholas J. Carriero, "Implementation of Tuple Space Machines", *UMI Dissertation Services*, Yale University, Dec. 1987.

[4] Nicholas Carriero and David Gelernter, "How to Write Parallel Programs", *The MIT Press*, Cambridge, Massachusetts, 1991.

[5] Nikos Drakos, "Ocean Models", *Computational Science Education Project*, U.S. Department of Energy, Aug. 1994.

[6] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck and Vaidy Sunderam, "PVM : Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing", *The MIT Press*, Cambridge, Massachusetts, 1994.

[7] Katherine S. Hedstrom, Institute of Marine and Coastal Sciences, Rutgers University, "User's Manual for a Semi-Spectral Primative Equation Ocean Circulation Model", Mar. 1994.

[8] Chris Hogue, "POWER Fortran Accelerator^{TM} User's Guide", *Silicon Graphics, Inc.*, Mountain View, CA, Jan. 1992.

[9] James J. O'Brien, "Advanced Physical Oceanographic Numerical Modeling", NATO ASI Series, Vol. 186, pp. 608.

[10] Germana Peggion, "Coupling a Basin-wide, Coarse Resolution North Atlantic and a Regional, Fine-resolution Gulf of Mexico Model", The University of Southern Mississippi Center for Ocean & Atmospheric Modeling, COAM Report (in preparation).

[11] B. R. Seyfarth, J. L. Bickham and M. R. Fernandez, "Glenda:An Environment for Easy Parallel Programming", *Scalable High Performance Computing Conference*, Knoxville, TN, May. 1994.

[12] B. R. Seyfarth, J. L. Bickham and Germana Peggion, "Glenda Software Design", The University of Southern Mississippi Center for Ocean & Atmospheric Modeling, COAM Technical Report, TR-3/95. Feb. 1995.

A The SWEM Experimental Region

Our experiments analyzed the response of the North Atlantic Ocean to the climatological monthly winds. The domain is configured in the reduced gravity formulation with an initial upper layer thickness of 450 meters. The numerical domain has closed boundaries. The boundary fitted coordinates are chosen to give a high spatial grid resolution (about 50 kilometers) in the western subtropical basin in order to analyze the circulation of the Caribbean Sea and its influence on the dynamics inside the Gulf of Mexico.

Figure 5 illustrates the annual mean upper layer thickness anomaly (in meters) from the numerical experiments. The solution indicates a double gyre system. The cyclonic (counterclockwise) subarctic gyre is unrealistic due to the close boundary configuration, but the western intensification of the anticyclonic (clockwise) gyre (the Gulf Stream) is well reproduced. The current separates from the coast in the proximity of Cape Hatteras and is dominated by large fluctuations with meanders and ring formation, as supported by observations and measurements.

The geometry of the western subtropical basin has a strong effect on the dynamics of the Caribbean Sea and Gulf of Mexico. Part of the return flow of the wind driven gyre enters the Caribbean Sea from several openings between the Antilles Islands. These branches organize into a narrow current (the Yucatan Current) which flows along the Mexican Coast, enters the Gulf of Mexico through the Yucatan Strait, and exits from the Florida Strait. There it rejoins the main return flow of the anticyclonic gyre [10].

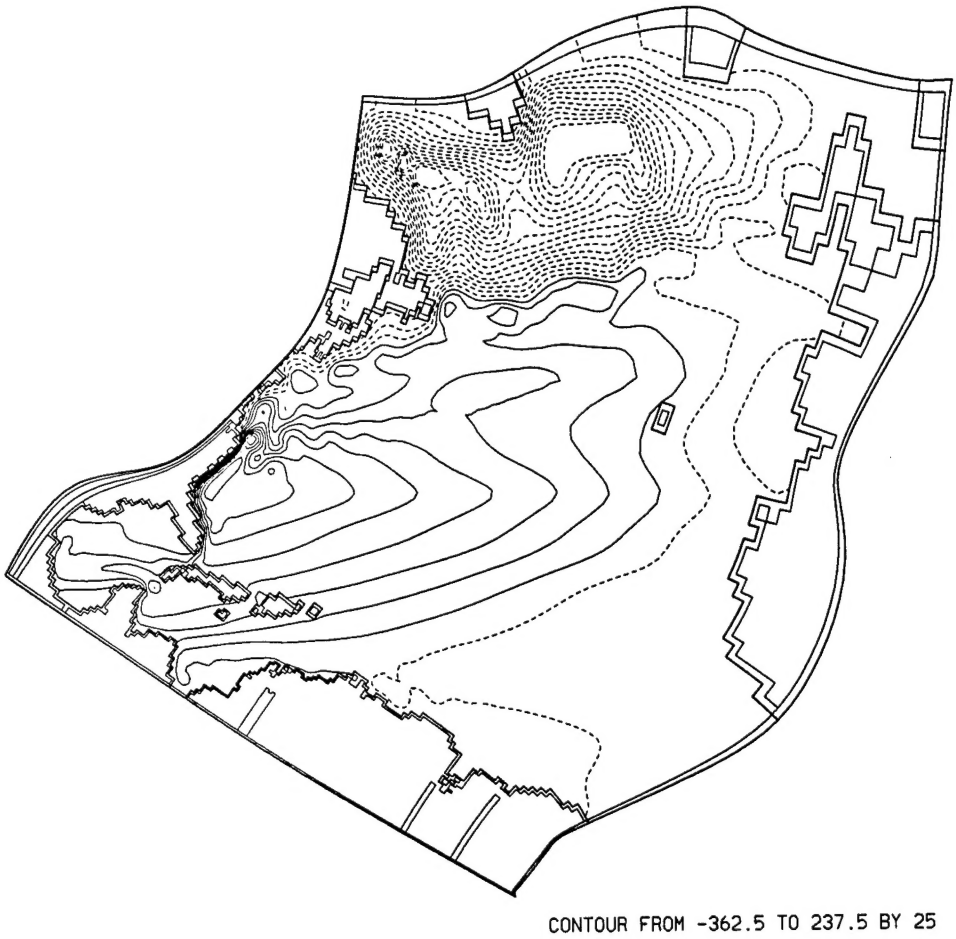


Figure 5: The annual mean sea surface displacement from the SWEM simulations.

B Parallel Version of SWEM

The parallel version of SWEM containing the SWEM FORTRAN code, as well as the files containing the Glenda subroutines ($master\_subs.cg$ and $worker\_subs.cg$), can be downloaded via anonymous ftp at

$$seabass.st.usm.edu$$

The two files containing the Glenda subroutines were added to the sequential SWEM code to facilitate the parallelism.

- The file $master\_subs.cg$ contains subroutines needed by the master program to communicate with the worker programs.
- The file $worker\_subs.cg$ contains subroutines needed by the worker programs to communicate with the master program.

All of the subroutines were written in C and used Glenda to handle the interprocess communication. Within the SWEM code, these subroutines are used as FORTRAN subroutines.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. Agency Use Only *(Leave blank).*	2. Report Date.	3. Report Type and Dates Covered.
	May 1995	Technical Report

4. Title and Subtitle.

PARALLEL OCEAN MODELING USING GLENDA

5. Funding Numbers.

Program Element No.

Project No.

Task No.

Accession No.

6. Author(s).

Jerry L. Bickham, II

7. Performing Organization Name(s) and Address(es).

Center for Ocean & Atmospheric Modeling
The University of Southern Mississippi
Building 1103, Room 249
Stennis Space Center, MS 39529-5005

8. Performing Organization Report Number.

TR-4/95

9. Sponsoring/Monitoring Agency Name(s) and Address(es).

Office of Naval Research
Code 1513: RKL
Ballston Centre Tower One
800 North Quincy Street
Arlington, VA 22217-5660

10. Sponsoring/Monitoring Agency Report Number.

11. Supplementary Notes.

ONR Research Grant No. N00014-92-J-4112

12a. Distribution/Availability Statement.

Approved for public release; distribution is unlimited.

12b. Distribution Code.

13. Abstract *(Maximum 200 words).*

In this document we describe how, with the use of Glenda, we were able to parallelize the ocean modeling program SWEM. After a description of Glenda and an introduction to the ocean model, we discuss how we parallelized the model. We first had to determine the array dependencies in the program in order to better understand the way we were going to handle data propagation. Once we had a grasp of how the data was to be propagated, there was a need to develop overlap definitions to help improve the readability of the code. But, we still encountered problems that had to be solved. Therefore, we had to use a wide assortment of debugging techniques to overcome these problems. With our problems finally solved and the program now complete, we tested our new version of SWEM and gathered results.

14. Subject Terms.

(U) PARALLEL PROCESSING, (U) GLENDA, (U) ARRAY DEPENDENCIES, (U) REDUCED GRAVITY OCEAN CIRCULATION MODEL

15. Number of Pages.

40

16. Price Code.

17. Security Classification of Report.	18. Security Classification of This Page.	19. Security Classification of Abstract.	20. Limitation of Abstract.
Unclassified	Unclassified	Unclassified	SAR